Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Slicing ATL Model Transformations for Scalable Deductive Verification and Fault Localization

Zheng Cheng[1], **Massimo Tisi**[2]

[1] Research Center INRIA Rennes - Bretagne Atlantique, Rennes, France
e-mail: zheng.cheng@inria.fr
[2] IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France
e-mail: massimo.tisi@imt-atlantique.fr

The date of receipt and acceptance will be inserted by the editor

Abstract. Model-driven engineering (MDE) is increasingly accepted in industry as an effective approach for managing the full life cycle of software development. In MDE, software models are manipulated, evolved and translated by model transformations (MT), up to code generation. Automatic deductive verification techniques have been proposed to guarantee that transformations satisfy correctness requirements (encoded as transformation contracts). However, to be transferable to industry, these techniques need to be scalable and provide the user with easily accessible feedback.

In MT-specific languages like ATL, we are able to infer static trace information (i.e. mappings among types of generated elements and rules that potentially generate these types). In this paper we show that this information can be used to decompose the MT contract and, for each sub-contract, slice the MT to the only rules that may be responsible for fulfilling it. Based on this contribution, we design a fault localization approach for MT, and a technique to significantly enhance scalability when verifying large MTs against a large number of contracts. We implement both these algorithms as extensions of the VeriATL verification system, and we show by experimentation that they increase its industry-readiness.

1 Introduction

Model-Driven Engineering (MDE), i.e. software engineering centered on software models, is widely recognized as an effective way to manage the complexity of software development. In MDE, software models are manipulated, evolved and translated by model transformation (MTs), up to code generation. An incorrect MT would generate faulty models, whose effect could be unpredictably propagated into subsequent MDE steps (e.g. code generation), and compromise the reliability of the whole software development process.

Deductive verification emphasizes the use of logic (e.g. Hoare logic [23]) to formally specify and prove program correctness. Due to the advancements in the last couple of decades in the performance of constraint solvers (especially satisfiability modulo theory - SMT), many researchers are interested in developing techniques that can partially or fully automate the deductive verification for the correctness of MTs (we refer the reader to [1] for an overview).

While industrial MTs are increasing in size and complexity (e.g. automotive industry [38],

medical data processing [42], aviation [6]), existing deductive verification approaches and tools show limitations that hinder their practical application.

Scalability is one of the major limitations. Current deductive verification tools do not provide clear evidence of their efficiency for large-scale MTs with a large number of rules and contracts [1]. Consequently, users may suffer from unbearably slow response when verification tasks scale. For example, as we show in our evaluation, the verification of a realistic refactoring MT with about 200 rules against 50 invariants takes hours (Section 6). In [8], the author argues that this lack of scalable techniques becomes one of the major reasons hampering the usage of verification in industrial MDE.

Another key issue is that, when the verification fails, the output of verification tools is often not easily exploitable for identifying and fixing the fault. In particular, industrial MDE users do not have the necessary background to be able to exploit the verifier feedback. Ideally, one of the most user-friendly solutions would be the introduction of fault localization techniques [37, 44], in order to directly point to the part of MT code that is responsible for the fault. Current deductive verification systems for MT have no support for fault localization. Consequently, manually examining the full MT and its contracts, and reasoning on the implicit rule interactions remains a complex and time-consuming routine to debug MTs.

In [13], we developed the *VeriATL* verification system to deductively verify the correctness of MTs written in the ATL language [27], w.r.t. given contracts (in terms of pre-/post-conditions). Like several other MT languages, ATL has a relational nature, i.e. its core aspect is a set of so-called matched rules, that describe the mappings between the elements in the source and target model. VeriATL automatically translates the axiomatic semantics of a given ATL transformation in the Boogie intermediate verification language [4], combined with a formal encoding of EMF metamodels [39] and OCL contracts. The Z3 automatic theorem prover [31] is then used by Boogie to verify the correctness of the ATL transformation. While the usefulness of VeriATL has been shown by experimentation [13], its original design suffers from the two mentioned limitations, i.e. it does not scale well, and does not provide accessible feedback to identify and fix the fault.

In this article, we argue that the relational nature of ATL can be exploited to address both the identified limitations. Thanks to the relational structure, we are able to deduce static trace information (i.e. inferred information among types of generated target elements and the rules that potentially generate these types) from ATL MTs. Then, we use this information to propose a **slicing** approach that first decomposes the postcondition of the MT into sub-goals, and for each sub-goal, slices out of the MT all the rules that do not impact the sub-goal. Specifically,

- First, we propose a set of sound natural deduction rules. The set includes 4 rules that are specific to the ATL language (based on the concept of static trace information), and 16 ordinary natural deduction rules for propositional and predicate logic [25]. Then, we propose an automated proof strategy that applies these deduction rules on the input OCL postcondition to generate sub-goals. Each sub-goal contains a list of newly deduced hypotheses, and aims to prove a sub-case of the input postcondition.
- Second, we exploit the hypotheses of each sub-goal to slice the ATL MT into a simpler transformation context that is specific to each sub-goal.

Finally we propose two solutions that apply our MT slicing technique to the tasks of enabling fault localization and enhancing scalability:

- **Fault Localization.** We apply our natural deduction rules to decompose each unverified postcondition in sub-goals, and generate several verification conditions (VCs), i.e. one for each generated sub-goal and corresponding MT slice. Then, we verify these new VCs, and present the user with

the unverified ones. The unverified sub-goals help the user pinpoint the fault in two ways: (a) the failing slice is underlined in the original MT code to help localizing the bug; (b) a set of debugging clues, deduced from the input postcondition are presented to alleviate the cognitive load for dealing with unverified sub-cases. The approach is evaluated by mutation analysis.

– **Scalability.** Before verifying each postcondition, we apply our slicing approach to slice the ATL MT into a simpler transformation context, thereby reducing the verification complexity/time of each postcondition (Section 5.1). We prove the correctness of the approach. Then we design and prove a grouping algorithm, to identify the postconditions that have high probability of sharing proofs when verified in a single verification task (Section 5.2). Our evaluation confirms that our approach improves verification performance up to an order of magnitude (79% in our use case) when the verification tasks of a MT are scaling up (Section 6).

These two solutions are implemented by extending VeriATL. The source code of our implementations and complete artifacts used in our evaluation are publicly available[1][2].

This paper extends an article contributed to the FASE 2017 conference [15] by the same authors. While the conference article was introducing the fault localization approach, this paper recognizes that the applicability of our slicing approach is more general, and can benefit other requirements for industry transfer, such as scalability.

Paper organization. Section 2 motivates by example the need for fault localization and scalability in MT verification. Section 3 presents our solution for fault localization in the deductive verification of MT. Section 4 illustrates in detail the key component of this first application, the deductive decomposition and slicing approach. Section 5 applies this slicing approach to our second task, i.e. enhancing general scalability in deductive MT verification. The practical applicability and performance of our solutions are shown by evaluation in Section 6. Finally, Section 7 compares our work with related research, and Section 8 presents our conclusions and proposed future work.

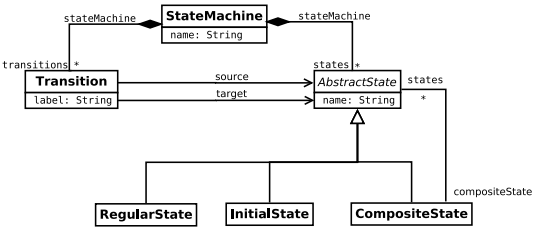


Fig. 1. The hierarchical and flattened state machine metamodel

2 Motivating Example

We consider as running case a MT that transforms hierarchical state machine (*HSM*) models to flattened state machine (*FSM*) models, namely the *HSM2FSM* transformation. Both models conform to the same simplified state machine metamodel (Fig. 1). For clarity, classifiers in the two metamodels are distinguished by the *HSM* and *FSM* prefix. In detail, a named *StateMachine* contains a set of labelled *Transitions* and named *AbstractStates*. Each *AbstractState* has a concrete type, which is either *RegularState*, *InitialState* or *CompositeState*. A *Transition* links a *source* to a *target AbstractState*. Moreover, *CompositeStates* are only allowed in the models of HSM, and optionally contain a set of *AbstractStates*.

Fig. 2 depicts a *HSM* model that includes a composite state[3]. Fig. 3 demonstrates how the *HSM2FSM* transformation is expected to flatten it: (a) composite states need to be removed, the initial state within needs to become a regular state, and all the other states

[1] A deductive approach for fault localization in ATL MTs (Online). `https://github.com/veriatl/VeriATL/tree/FaultLoc`.

[2] On scalability of deductive verification for ATL MTs (Online). `https://github.com/veriatl/VeriATL/tree/Scalability`.

[3] We name the initial states in the concrete syntax of *HSM* and *FSM* models for readability.

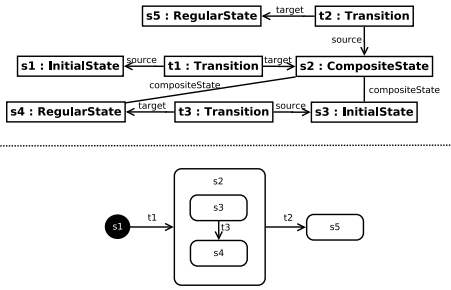


Fig. 2. Example of HSM. Abstract (top) and concrete graphical syntax (bottom)

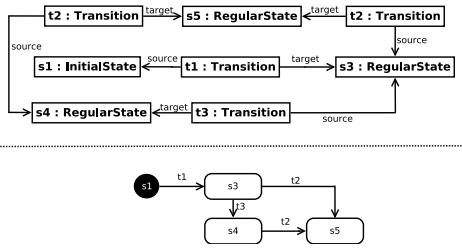


Fig. 3. FSM obtained by flattening the HSM in Fig. 2. Abstract (top) and concrete syntax (bottom)

need to be preserved; (b) transitions targeting a composite state need to redirect to the initial state of such composite state, transitions outgoing from a composite state need to be duplicated for the states within such composite state, and all the other transitions need to be preserved.

Specifying OCL contracts. We consider a contract-based development scenario where the developer first specifies correctness conditions for the to-be-developed ATL transformation by using OCL contracts. For example, let us consider the contract shown in Listing 1. The precondition *Pre1* specifies that in the input model, each *Transition* has at least one *source*. The postcondition *Post1* specifies that in the output model, each *Transition* has at least one *source*.

While pre-/post-conditions in Listing 1 are generic well-formedness properties for state machines, the user could specify transformation-specific properties in the same way. For instance, the complete version of this use case also contains the following transformation-specific

```
1 context HSM!Transition inv Pre1:
2   HSM!Transition.allInstances()->forAll(t | not
        t.source.oclIsUndefined())
3 ---------------------------------
4 context FSM!Transition inv Post1:
5   FSM!Transition.allInstances()->forAll(t | not
        t.source.oclIsUndefined())
```

Listing 1. The OCL contracts for HSM and FSM

contract: if states have unique names within any source model, states will have unique names also in the generated target model. In general, there are no restrictions on what kind of correctness conditions could be expressed, as long as they are expressed in the subset of OCL we considered in this work (see language support in Section 6.3 for more details).

Developing the ATL transformation. Then, the developer implements the ATL transformation *HSM2FSM* (a snippet is shown in Listing 2[4]). The transformation is defined via a list of ATL matched rules in a mapping style. The first rule maps each *StateMachine* element to the output model (*SM2SM*). Then, we have two rules to transform *AbstractStates*: regular states are preserved (*RS2RS*), initial states are transformed into regular states when they are within a composite state (*IS2RS*). Notice here that initial states are deliberately transformed partially to demonstrate our problem, i.e. we miss a rule that specifies how to transform initial states when they are **not** within a composite state. The remaining three rules are responsible for mapping the *Transitions* of the input state machine.

Each ATL matched rule has a *from* section where the source pattern to be matched in the source model is specified. An optional OCL constraint may be added as the guard, and a rule is applicable only if the guard evaluates to true on the source pattern. Each rule also has a *to* section which specifies the elements to be created in the target model. The rule initializes the attributes/associations of a generated target element via the binding operator (`<-`). An important feature of ATL is the use of an

[4] Our HSM2FSM transformation is adapted from [10]. The full version can be accessed at: `https://goo.gl/MbwiJC`.

```
1  module HSM2FSM;
2  create OUT : FSM from IN : HSM;
3
4  rule SM2SM {
5    from sm1 : HSM!StateMachine
6    to sm2 : FSM!StateMachine
7      ( name <- sm1.name ) }
8
9  rule RS2RS {
10   from rs1 : HSM!RegularState
11   to rs2 : FSM!RegularState
12   ( stateMachine <- rs1.stateMachine,
13     name <- rs1.name ) }
14
15 rule IS2RS {
16   from is1 : HSM!InitialState
17     (not is1.compositeState.oclIsUndefined())
18   to   rs2 : FSM!RegularState
19     ( stateMachine <- is1.stateMachine,
20       name <- is1.name ) }
21
22 -- mapping each transition between two
       non-composite states
23 rule T2TA { ... }
24
25 -- mapping each transition whose source is a
       composite state
26 rule T2TB { ... }
27
28 -- mapping each transition whose target is a
       composite state
29 rule T2TC {
30   from t1 : HSM!Transition,
31        src : HSM!AbstractState,
32        trg : HSM!CompositeState,
33        c : HSM!InitialState
34     ( t1.source = src and t1.target = trg
35       and c.compositeState = trg
36       and not src.oclIsTypeOf(HSM!CompositeState))
37   to   t2 : FSM!Transition
38     ( label <- t1.label,
39       stateMachine <- t1.stateMachine,
40       source <- src,
41       target <- c }
```

Listing 2. Snippet of the HSM2FSM MT in ATL

implicit *resolution* algorithm during the target property initialization. Here we illustrate the algorithm by an example: 1) considering the binding *stateMachine* `<-` *rs1.stateMachine* in the *RS2RS* rule (line 13 of Listing 2), its right-hand side is evaluated to be a source element of type *HSM!StateMachine*; 2) the resolution algorithm then resolves such source element to its corresponding target element of type *FSM!StateMachine* (generated by the *SM2SM* rule); 3) the resolved result is assigned to the left-hand side of the binding. While not strictly needed for understanding this paper, we refer the reader to [27] for a full description of the ATL language.

Formally verifying the ATL transformation by VeriATL. The source and target EMF metamodels and OCL contracts combined with the developed ATL transformation form a VC which can be used to verify the correctness of the ATL transformation for all possible inputs, i.e. MM, Pre, Exec $\vdash$ Post. The VC semantically means that, assuming the axiomatic semantics of the involved EMF metamodels (*MM*) and OCL preconditions (*Pre*), by executing the developed ATL transformation (*Exec*), the specified OCL postcondition has to hold (*Post*).

In previous work, Cheng et al. have developed the VeriATL verification system that allows such VCs to be soundly verified [13]. Specifically, the VeriATL system describes in Boogie what correctness means for the ATL language in terms of structural VCs. Then, VeriATL delegates the task of interacting with Z3 for proving these VCs to Boogie.In particular, VeriATL encodes: 1) *MM* using axiomatized Boogie constants to capture the semantics of metamodel classifiers and structural features, 2) *Pre* and *Post* using first order logic Boogie assumption and assertion statements respectively to capture the pre-/post- conditions of MTs, 3) *Exec* using Boogie procedures to capture the matching and applying semantics of ATL MTs. We refer our previous work [13] for the technical description of how to map a VC to its corresponding Boogie program.

Problem 1: Debugging. In our example, VeriATL successfully reports that the OCL postcondition *Post1* is not verified by the MT in Listing 2. This means that the transformation does not guarantee that each *Transition* has at least one *source* in the output model. Without any capability of fault localization, the developer needs to manually inspect the full transformation and contracts to understand that the transformation is incorrect because of the absence of an ATL rule to transform *InitialStates* that are not within a *CompositeState*.

To address problem 1, our aim is to design a fault localization approach that automatically presents users with the information in Listing 3 (described in detail in the follow-

ing Section 3.1). The output includes: (a) the slice of the MT code containing the bug (that in this case involves only three rules), (b) a set of debugging clues, deduced from the original postcondition (in this case pointing to the the fact that T2TC can generate transitions without source). We argue that this information is a valuable help in identifying the cause of the bug.

Problem 2: Scalability. While for illustrative purposes in this paper we consider a very small transformation, it is not difficult to extend it to a realistically sized scenario. For instance we can imagine Listing 2 to be part (up to renaming) of a refactoring transformation for the full UML (e.g. including statecharts, but also class diagrams, sequence diagrams, activity diagrams etc.). Since the UML v2.5 [33] metamodel contains 194 concrete classifiers (plus 70 abstract classifiers), even the basic task of simply copying all the elements not involved in the refactoring of Listing 2 would require at least 194 rules. Such large transformation would need to be verified against the full set of UML invariants, that describe the well-formedness of UML artifacts according to the specification[5]. While standard VeriATL is successfully used for contract-based development of smaller transformations [15], in our experimentation we show that it needs hours to verify a refactoring on the full UML against 50 invariants.

To address problem 2, we design a scalable verification approach aiming at 1) reducing the verification complexity/time of each postcondition (Section 5.1) and 2) grouping postconditions that have high probability of sharing proofs when verified in a single verification task (Section 5.2). Thanks to these techniques the verification time of our use case in UML refactoring is reduced by about 79%.

[5] OCL invariants for UML. `http://bit.ly/UMLContracts`

```
1  context HSM!Transition inv Pre1: ...
2
3  rule RS2RS { ... }
4  rule IS2RS { ... }
5  rule T2TC { ... }
6
7  context FSM!Transition inv Post1_sub:
8  *hypothesis* var t0
9  *hypothesis*
       FSM!Transition.allInstances()->includes(t0)
10 *hypothesis* genBy(t0,T2TC)
11 *hypothesis* t0.source.oclIsUndefined()
12 *hypothesis* not (genBy(t0.source,RS2RS) or
       genBy(t0.source,IS2RS))
13 *goal* false
```

Listing 3. The problematic transformation scenario of the *HSM2FSM* transformation w.r.t. *Post1*

3 Fault Localization for Model Transformation

3.1 Fault localization in the running case

We propose a fault localization approach that, in our running example, presents the user with two problematic transformation scenarios. One of them is shown in Listing 3. The scenario consists of the input preconditions (abbreviated at line 1), a slice of the transformation (abbreviated at lines 3 - 5), and a sub-goal derived from the input postcondition. The subgoal contains a list of hypotheses (lines 7 - 12) with a conclusion (line 13).

The scenario in Listing 3 contains the following information, that we believe to be valuable in identifying and fixing the fault:

- *Transformation slice.* The only relevant rules for the fault captured by this problematic transformation scenario are *RS2RS*, *IS2RS* and *T2TC* (lines 3 - 5). They can be directly highlighted in the source code editor.
- *Debugging clues.* The error occurs when a transition *t0* is generated by the rule *T2TC* (lines 8 - 10), and when the *source* state of the transition is not generated (line 11). In addition, the absence of the *source* for *t0* is due to the fact that none of the *RS2RS* and *IS2RS* rules is invoked to generate it (line 12).

From this information, the user could find a counter-example in the source models that

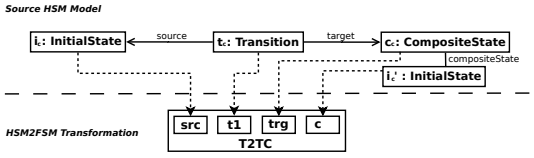


Fig. 4. Counter-example derived from Listing 3 that falsify *Post1*

falsifies *Post1* (shown in the top of Fig. 4): a transition t_c between an initial state i_c (which is not within a composite state) and a composite state c_c, where c_c composites another initial state $i_c{}'$. This counter-example matches the source pattern of the *T2TC* rule (as shown in the bottom of Fig. 4). However, when the *T2TC* rule tries to initialize the *source* of the generated transition *t2* (line 41 in Listing 2), i_c cannot be resolved because there is no rule to match it. In this case, i_c (of type *HSM!InitialState*)) is directly used to initialize the *source* of *t2* (*t2.source* is expected to be a sub-type of *FSM!AbstractState*). This causes an exception of type mismatch, thus falsifying *Post1*. The other problematic transformation scenario pinpoints the same fault, showing that *Post1* is not verified by the MT also when *t0* is generated by *T2TA*.

In the next sections, we describe in detail how we automatically generate problematic transformation scenarios like the one shown in Listing 3.

3.2 Solution overview

The flowchart in Fig. 5 shows a bird's eye view of our approach to enable fault localization for VeriATL. The process takes the involved metamodels, all the OCL preconditions, the ATL transformation and one of the OCL postconditions as inputs. We require all inputs to be syntactically correct. If VeriATL successfully verifies the input ATL transformation, we directly report a confirmation message to indicate its correctness (w.r.t. the given postcondition) and the process ends. Otherwise, we generate a set of problematic transformation scenarios, and a proof tree to the transformation developer.

To generate problematic transformation scenarios, we first perform a systematic approach to generate sub-goals for the input OCL postcondition. Our approach is based on a set of sound natural deduction rules (Section 4.1). The set contains 16 rules for propositional and predicate logic (such as introduction/elimination rules for $\wedge$ and $\vee$ [25]), but also 4 rules specifically designed for ATL expressions (e.g. rewriting single-valued navigation expression).

Then, we design an automated proof strategy that applies these natural deduction rules on the input OCL postcondition (Section 4.2). Executing our proof strategy generates a proof tree. The non-leaf nodes are intermediate results of deduction rule applications. The leafs in the tree are the sub-goals to prove. Each sub-goal consists of a list of hypotheses and a conclusion to be verified. The aim of our automated proof strategy is to simplify the original postcondition as much as possible to obtain a set of sub-conclusions to prove. As a by-product, we also deduce new hypotheses from the input postcondition and the transformation, as debugging clues.

Next, we use the trace information in the hypotheses of each sub-goal to slice the input MT into simpler transformation contexts (Section 4.3). We then form a new VC for each sub-goal consisting of the semantics of metamodels, input OCL preconditions, sliced transformation context, its hypotheses and its conclusion.

We send these new VCs to the VeriATL verification system to check. Notice that successfully proving these new VCs implies the satisfaction of the input OCL postcondition. If any of these new VCs is not verified by VeriATL, the input OCL preconditions, the corresponding sliced transformation context, hypotheses and conclusion of the VC are presented to the user as a problematic transformation scenario for fault localization. The VCs that were automatically proved by VeriATL are pruned away, and are not presented to the transformation developer. This deductive verification step by VeriATL makes the whole process practical, since the user is presented with a limited number of meaningful scenarios.

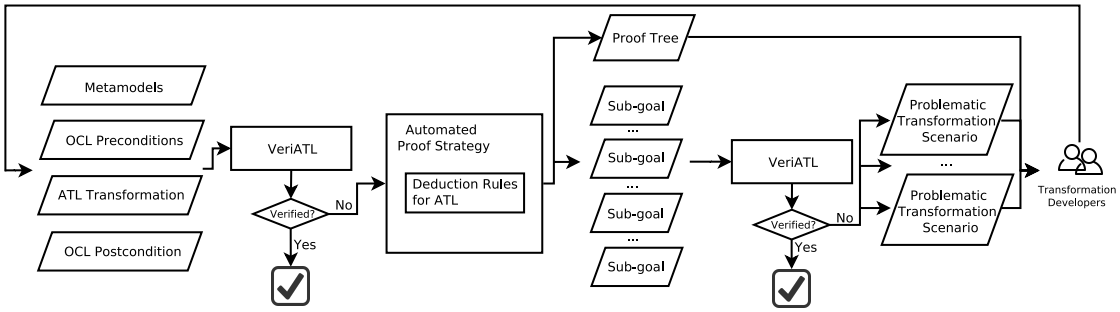


Fig. 5. Overview of providing fault localization for VeriATL

Then, the transformation developer consults the generated problematic transformation scenarios and the proof tree to debug the ATL transformation. If modifications are made on the inputs to fix the bug, the generation of sub-goals needs to start over. The whole process keeps iterating until the input ATL transformation is correct w.r.t. the input OCL postcondition.

4 A Deductive Approach to Transformation Slicing

The key step in the solution for fault localization that we described in the previous section is a general technique for: 1) decomposing the postcondition into sub-goals by applying MT-specific natural deduction rules, and 2) for each sub-goal, slice the MT to the only rules that may be responsible for fulfilling that sub-goal.

In this section we describe this algorithm in detail, and in the next section we show that its usefulness goes beyond fault localization, by applying it for enhancing the general scalability of VeriATL.

4.1 Natural Deduction Rules for ATL

Our approach relies on 20 natural deduction rules (7 introduction rules and 13 elimination rules). The 4 elimination rules (abbreviated by X_e) that specifically involve ATL are shown in Fig. 6. The other rules are common natural deduction rules for propositional and predicate logic [25]. Regarding the notations in our natural deduction rules:

- Each rule has a list of hypotheses and a conclusion, separated by a line. We use standard notation for typing (:) and set operations.
- Some special notations in the rules are T for a type, MM_{T} for the target metamodel, R_n for a rule n in the input ATL transformation, $x.a$ for a navigation expression, and i for a fresh variable / model element. In addition, we introduce the following auxiliary functions: *cl* returns the classifier types of the given metamodel, *trace* returns the ATL rules that generate the input type (i.e. the static trace information)[6], *genBy(i,R)* is a predicate to indicate that a model element i is generated by the rule R, *unDef(i)* abbreviates *i.oclIsUndefined()*, and *All(T)* abbreviates *T.allInstances()*.

Some explanation is in order for the natural deduction rules that are specific to ATL:

- First, we have two type elimination rules (TP_{e1}, TP_{e2}). TP_{e1} states that every single-valued navigation expression of the type T in the target metamodel is either a member of all generated instances of type T or undefined. TP_{e2} states that the cardinality of every multi-valued navigation expression of the type T in the target metamodel is either greater than zero (and every element i in the multi-valued naviga-

[6] In practice, we fill in the *trace* function by examining the output element types of each ATL rule, i.e. the *to* section of each rule.

$$\frac{x.a : T \qquad T \in cl(MM_{\mathrm{T}})}{x.a \in All(T) \vee unDef(x.a)}\ \mathrm{TP}_{e1}$$

$$\frac{x.a : Seq\ T \qquad T \in cl(MM_{\mathrm{T}})}{(|x.a| > 0 \wedge \forall i \cdot (i \in x.a \Rightarrow i \in All(T) \vee unDef(i))) \vee |x.a| = 0}\ \mathrm{TP}_{e2}$$

$$\frac{T \in cl(MM_{\mathrm{T}}) \qquad trace(T) = \{R_1, \ldots, R_n\} \qquad i \in All(T)}{genBy(i, R_1) \vee \ldots \vee genBy(i, R_n)}\ \mathrm{TR}_{e1}$$

$$\frac{T \in cl(MM_{\mathrm{T}}) \qquad trace(T) = \{R_1, \ldots, R_n\} \qquad i : T \qquad unDef(i)}{\neg(genBy(i, R_1) \vee \ldots \vee genBy(i, R_n))}\ \mathrm{TR}_{e2}$$

Fig. 6. Natural deduction rules that are specific to ATL

tion expression is either a member of all generated instances of type T or undefined) or equal to zero.

- Second, we have 2 elimination rules for trace (TR_{e1}, TR_{e2}). These rules state that, given that the rules R_1,...,R_n in the input ATL transformation are responsible to create model elements of type T in the target metamodel, we may rightfully conclude that:
  - (TR_{e1}): every created element i of type T is generated by one of the rules R_1,...,R_n.
  - (TR_{e2}): every undefined element i of type T is not generated by any of the rules R_1,...,R_n.

The set of natural deduction rules is sound, as we show in the rest of this section. However, it is not complete, and we expect to extend it in future work. As detailed in Section 6.3, when the bug affects a postcondition that we don't support because of this incompleteness, we report to the user our inability to perform fault localization for that postcondition.

Soundness of natural deduction rules. The soundness of our natural deduction rules is based on the operational semantics of the ATL language. Specifically, the soundness for type elimination rules TP_{e1} and TP_{e2} is straightforward. We prove their soundness by enumerating the possible states of initialized navigation expressions for target elements. Specifically, assuming that the state of a navigation expression $x.a$ is initialized in the form $x.a$<-exp where $x.a$ is of a non-primitive type T:

- If *exp* is not a collection type and cannot be resolved (i.e. *exp* cannot match the source pattern of any ATL rules), then $x.a$ is *undefined*[7].
- If *exp* is not a collection type and can be resolved, then the generated target element of the ATL rule that matches *exp* is assigned to $x.a$. Consequently, $x.a$ could be either a member of *All(T)* (when the resolution result is of type T) or undefined (when it is not).
- If *exp* is of collection type, then all of the elements in *exp* are resolved individually, and the resolved results are put together into a pre-allocated collection *col*, and *col* is assigned to $x.a$.

The first two cases explain the two possible states of every single-valued navigation expressions (TP_{e1}). The third case explains the two possible states of every multi-valued navigation expressions (TP_{e2}).

The soundness of trace elimination rules TR_{e1} is based on the surjectivity between each ATL rule and the type of its created target elements [10]: elements in the target metamodel exist if they have been created by an ATL rule since standard ATL transformations are always executed on an initially empty target model. When a type can be generated by executing more than one rule, then a disjunction considering all these possibilities is made for every generated elements of this type.

About the soundness of the TR_{e2} rule, we observe that if a target element of type T is undefined, then clearly it does not belong to *All(T)*. In addition, the operational semantics for the ATL language specifies that if a rule R is specified to generate elements of type T, then every target elements of type T generated by that rule belong to *All(T)* (i.e. $R \in trace(T) \Rightarrow \forall i \cdot (genBy(i, R) \Rightarrow i \in All(T))$) [13]. Thus, TR_{e2} is sound as a logical consequence of the operational semantics for the

[7] In fact, the value of *exp* is assigned to $x.a$ because of resolution failure. This causes a type mismatch exception and results in the value of $x.a$ becoming undefined (we consider ATL transformations in non-refinement mode where the source and target metamodels are different).

ATL language (i.e. $R \in trace(T) \Rightarrow \forall i \cdot (i \notin All(T) \Rightarrow \neg genBy(i, R))$).

4.2 Automated Proof Strategy

A proof strategy is a sequence of proof steps. Each step defines the consequences of applying a natural deduction rule on a proof tree. A proof tree consists of a set of nodes. Each node is constructed by a set of OCL expressions as hypotheses, an OCL expression as the conclusion, and another node as its parents node.

Next, we illustrate a proof strategy (Algorithm 1) that automatically applies our natural deduction rules on the input OCL postcondition. The goal is to automate the derivation of information from the postcondition as hypotheses, and simplify the postcondition as much as possible.

Algorithm 1 An automated proof strategy for VeriATL

```
1: Tree ← {createNode({}, Post, null)}
2: do
3:    leafs ← size(getLeafs(Tree))
4:    for each node leaf ∈ getLeafs(Tree) do
5:       Tree ← intro(leaf) ∪ Tree
6:    end for
7: while leafs ≠ size(getLeafs(Tree))
8: do
9:    leafs ← size(getLeafs(Tree))
10:   for each node leaf ∈ getLeafs(Tree) do
11:      Tree ← elimin(leaf) ∪ Tree
12:   end for
13: while leafs ≠ size(getLeafs(Tree))
```

Our proof strategy takes one argument which is one of the input postconditions. Then, it initializes the proof tree by constructing a new root node of the input postcondition as conclusion and no hypotheses and no parent node (line 1). Next, our proof strategy takes two sequences of proof steps. The first sequence applies the *introduction* rules on the leaf nodes of the proof tree to generate new leafs (lines 2 - 7). It terminates when no new leafs are yield (line 7). The second sequence of steps applies the *elimination* rules on the leaf nodes of the proof tree (lines 8 - 13). We only apply type elimination rules on a leaf when: (a) a free variable is in its hypotheses, and (b) a navigation expression of the free variable is referred by its hypotheses. Furthermore, to ensure termination, we enforce that if applying a rule on a node does not yield new descendants (i.e. whose hypotheses or conclusion are different from their parent), then we do not attach new nodes to the proof tree.

4.3 Transformation Slicing

Executing our proof strategy generates a proof tree. The leafs in the tree are the sub-goals to prove by VeriATL. Next, we use the rules referred by the *genBy* predicates in the hypotheses of each sub-goal to slice the input MT into a simpler transformation context. We then form a new VC for each sub-goal consisting of the axiomatic semantics of metamodels, input OCL preconditions, sliced transformation context ($Exec_{sliced}$), its hypotheses and its conclusion, i.e. MM, Pre, $Exec_{sliced}$, Hypotheses $\vdash$ Conclusion.

If any of these new VCs is not verified by VeriATL, the input OCL preconditions, the corresponding sliced transformation context, hypotheses and conclusion of the VC are constructed as a problematic transformation scenario to report back to the user for fault localization (as shown in Listing 3).

Correctness. The correctness of our transformation slicing is based on the concept of rule irrelevance (Theorem 1). That is the axiomatic semantics of the rule(s) being sliced away ($Exec_{irrelevant}$) has no effects to the verification outcome of its sub-goal.

Theorem 1 (Rule Irrelevance - Sub-goals). *MM, Pre,* $Exec_{sliced}$*, Hypotheses* $\vdash$ *Conclusion* $\iff$ *MM, Pre,* $Exec_{sliced \cup irrelevant}$*, Hypotheses* $\vdash$ *Conclusion* [8]

Proof. Each ATL rule is exclusively responsible for the generation of its output elements

[8] $Exec_{sliced \cup irrelevant} \iff Exec_{sliced} \wedge Exec_{irrelevant}$

(i.e. no aliasing) [22, 41]. Hence, when a sub-goal specifies a condition that a set of target elements should satisfy, the rules that do not generate these elements have no effects to the verification outcome of its sub-goal. These rules can hence be safely sliced away.

5 Scalability by Transformation Slicing

Being able to decompose contracts and slice the transformation as described in the previous section, can be also exploited internally for enhancing the scalability of the verification process.

Typically, verification tools like VeriATL will first formulate VCs to pass to the theorem prover. Then, they may try to enhance performance by decomposing and/or composing these VCs:

- VCs can be decomposed, creating smaller VCs that may be more manageable for the theorem prover. For instance Leino et al. introduce a VC optimization in Boogie (hereby referred as *VC splitting*) to automatically split VCs based on the control-flow information of programs [30]. The idea is to align each postcondition to its corresponding path(s) in the control flow, then to form smaller VCs to be verified in parallel.
- VCs can be composed, e.g. by constructing a single VC to prove the conjunction of all postconditions (hereby referred as *VC conjunction*). This has the benefit to enable sharing parts of the proofs of different postconditions (i.e. the theorem prover might discover that lemmas for proving a conjunct are also useful for proving other terms).

However, domain-agnostic composition or decomposition does not provide significant speed-ups to our running case. For instance the Boogie-level VC splitting has no measurable effect. Once the transformation is translated in an imperative Boogie program, transformation rules, even if independent from each other, become part of a single path in the control-flow [13]. Hence, each postcondition is always aligned to the whole set of transformation rules. We argue that a similar behavior would have been observed also if the transformation was directly developed in an imperative language (Boogie or a general-purpose language): a domain-agnostic VC optimization does not have enough information to identify the independent computation units within the transformation (i.e. the rules).

In what follows, we propose a two-step method to construct more efficient VCs for verifying large MTs. In the first step, we want to apply our MT-specific slicing technique (Section 4) on top of the Boogie-level VC splitting (Section 5.1): thanks to the abstraction level of the ATL language, we can align each postcondition to the ATL rules it depends on, thereby greatly reduce the size of each constructed VC. In the second step, we propose an ATL-specific algorithm to decide when to conjunct or split VCs (Section 5.2), improving on domain-agnostic VC conjunction.

5.1 Applying the Slicing Approach

Our first ATL-level optimization aims to verify each postcondition only against the rules that may impact it (instead of verifying it against the full MT), thus reducing the burden on the SMT solver.

This is achieved by a transformation slicing approach for postconditions: first applying the decomposition in sub-goals and the slicing technique from Section 4, and then merging the slices of the generated sub-goals. The MT rules that lay outside the union are sliced away, and the VC for each postcondition becomes: MM, Pre, $\text{Exec}_{slice} \vdash$ Post, where Exec_{slice} stands for axiomatic semantics of sliced transformation, and the sliced transformation is the union of the rules that affect the sub-goals of each postcondition.

Correctness. We first define a complete application of the automated proof strategy in Definition 1.

Definition 1 (Complete application of the automated proof strategy). The automated

proof strategy is completely applied to a postcondition if it correctly identifies every element of target types referred by each sub-goal and every rule that may generate them.

Clearly, if not detected, an incomplete application of our automated proof strategy could cause our transformation slicing to erroneously slice away the rules that a postcondition might depend on, and invalidate our slicing approach to verify postconditions. We will discuss how we currently handle and can improve the completeness of the automated proof strategy in Section 6.3. One of the keys in handling incomplete cases is that we defensively construct the slice to be the full MT. Thus, the VCs of the incomplete cases become MM, Pre, Exec $\vdash$ Post. This key point is used to establish the correctness of our slicing approach to verify postconditions (Theorem 2).

Theorem 2 (Rule Irrelevance - Postconditions). *MM, Pre,* $Exec_{sliced}$, $\vdash$ *Post* $\iff$ *MM, Pre,* $Exec_{sliced \cup irrelevant}$ $\vdash$ *Post*

Proof. We prove this theorem by a case analysis on whether the application of our automated proof strategy is complete:

- Assuming our automated proof strategy is completely applied. First, because the soundness of our natural deduction rules, it guarantees the generated sub-goals are a sound abstraction of their corresponding original postcondition. Second, based on the assumption that our automated proof strategy is completely applied, we can ensure that the union of the static trace information for each sub-goal of a postcondition contains all the rules that might affect the verification result of such postcondition. Based on the previous two points, we can conclude that slicing away its irrelevant rules has no effects to the verification outcome of a postcondition following the same proof strategy as in Theorem 1.
- Assuming our automated proof strategy is not completely applied. In this case, we will defensively use the full transformation as the slice, then in this case, our theorem becomes MM, Pre, Exec $\vdash$ Post $\iff$ MM, Pre, Exec $\vdash$ Post, which is trivially proved.

```
1 context HSM!Transition inv Pre1: ...
2
3 rule RS2RS { ... }
4 rule IS2RS { ... }
5 rule T2TA { ... }
6 rule T2TB { ... }
7 rule T2TC { ... }
8
9 context FSM!Transition inv Post1: ...
```

Listing 4. The optimized VC for *Post1* of Listing 1 constructed by program slicing

For example, Listing 4 shows the constructed VC for *Post1* of Listing 1 by using our program slicing technique. It concisely aligns *Post1* to 4 responsible rules in the UML refactoring transformation. Note that the same slice is obtained when the rules in Listing 2 are a part of a full UML refactoring. Its verification in our experimental setup (Section 6) requires less than 15 seconds, whereas verifying the same postcondition on the full transformation would exceed the 180s timeout.

5.2 Grouping VCs for Proof Sharing

After transformation slicing, we obtain a simpler VC for each postcondition. Now we aim to group the VCs obtained from the previous step in order to further improve performance. In particular, by detecting VCs that are related and grouping them in a conjunction, we encourage the underlying SMT solver to reuse subproofs while solving them. We propose an heuristics to identify the postconditions that should be compositionally verified, *by leveraging again the results from our deductive slicing approach.*

In our context, grouping of two VCs A and B means that MM, Pre, $\text{Exec}_{A \cup B} \vdash \text{Post}_A \wedge \text{Post}_B$. That is, taking account of the axiomatics semantics of metamodel, preconditions, and rules impacting A or B, the VC proves the conjunction of postconditions A and B.

It is difficult to precisely identify the cases in which grouping two VCs will improve efficiency. Our main idea is to prioritize groups that have high probability of sharing subproofs. Conservatively, we also want to avoid grouping an already complex VC with any other one,

but this requires to be able to estimate verification complexity. Moreover we want to base our algorithm exclusively on static information from VCs, because obtaining dynamic information is usually expensive in a large-scale MT settings.

We propose an algorithm based on two properties that are obtained by applying the natural deduction rules of our slicing approach: *number of static traces* and *number of sub-goals* for each postcondition. Intuitively, each one of the two properties is representative of a different cause of complexity: 1) when a postcondition is associated with a large number of static traces, its verification is challenging because it needs to consider a large part of the transformation, i.e. a large set of semantic axioms generated in Boogie by VeriATL; 2) a postcondition that results a large number of sub-goals, indicates a large number of combinations that the theorem prover will have to consider in a case analysis step.

We present our grouping approach in Algorithm 2. Its inputs are a set of postconditions P, and other two parameters: max traces per group (MAX_t) and max sub-goals per group (MAX_s). The result are VCs in groups (G).

The algorithm starts by sorting the input postconditions according to their trace set size (in ascending order). Then, for each postcondition p, it tries to pick from G the candidate groups (C) that may be grouped with p (lines 5 to 10). A group is considered to be a candidate group to host the given postcondition if the inclusion of the postcondition in the candidate group (*trail*) does not yield a group whose trace and sub-goals exceed MAX_t and MAX_s.

If there are no candidate groups to host the given postcondition, a new group is created (lines 11 to 12). Otherwise, we rank the suitability of candidate groups to host the postcondition by using the auxiliary function *rank* (lines 13 to 15). A group A has a higher rank than another group B to host a given postcondition p, if grouping A and p yields a smaller trace set than grouping B and p. When two groups are with the same ranking in terms of traces, we subsequently give a higher rank to the group that yields smaller total number of sub-goals when including the input postcondition.

This ranking is a key aspect of the grouping approach: (a) postconditions with overlapping trace sets are prioritized (since the union of their trace sets will be smaller). This raises the probability of proof sharing, since overlapping trace sets indicate that the proof of the two postconditions has to consider the logic of some common transformation rules. (b) postconditions with shared sub-goals are prioritized (since the union of total number of sub-goals will be smaller). This also raises the probability of proof sharing, since case analysis on the same sub-goals does not need to be analyzed again.

Finally, after each postcondition found a group in G that can host it, we generate VCs for each group in G and return them.

Algorithm 2 Algorithm for grouping VCs (P, MAX_t, MAX_s)

```
1:  P ← sort_t(P)
2:  G ← {}
3:  for each p ∈ P do
4:    C ← {}
5:    for each g ∈ G do
6:      trail ← group(g, p)
7:      if trail_t < MAX_t ∧ trail_s < MAX_s then
8:        C ← C + g
9:      end if
10:   end for
11:   if |C| = 0 then
12:     G ← G + {p}
13:   else
14:     G ← G + rank(C, p)
15:   end if
16: end for
17: return generate(G)
```

Note that the verification of a group of VCs yields a single result for the group. If the users wants to know exactly which postconditions have failed, they will need to verify the postconditions in the failed group separately.

Correctness. The correctness of our grouping algorithm is shown by its soundness as stated in Theorem 3.

Theorem 3 (Soundness of Grouping). *MM, Pre,* $Exec_{A \cup B} \vdash Post_A \wedge Post_B \implies$ *MM, Pre,* $Exec_A \vdash Post_A \wedge$ *MM, Pre,* $Exec_B \vdash Post_B$

Proof. Following the consequences of logical conjunction and Theorem 2.

6 Evaluation

In this section, we first evaluate the practical applicability of our fault localization approach (Section 6.1), then we assess the scalability of our performance optimizations (Section 6.2). Last, we conclude this section with a discussion of the obtained results and lessons learned (Section 6.3).

Our evaluation uses the VeriATL verification system [13], which is based on the Boogie verifier (version 2.3) and Z3 (version 4.5). The evaluation is performed on an Intel 3 GHz machine with 16 GB of memory running the Windows operating system. VeriATL encodes the axiomatic semantics of the ATL language (version 3.7). The automated proof strategy and its corresponding natural deduction rules are currently implemented in Java. We configure Boogie with the following arguments for fine-grained performance metrics:

- *timeout:180* (using a verification timeout of 180 seconds)
- *traceTimes* (using the internal Boogie API to calculate verification time).

6.1 Fault Localization Evaluation

Before diving into the details of evaluation results and analysis, we first formulate our research questions and describe the evaluation setup.

6.1.1 Research questions

We formulate two research questions to evaluate our fault localization approach:

(RQ1) Can our approach **correctly** pinpoint the faults in the given MT?

(RQ2) Can our approach **efficiently** pinpoint the faults in the given MT?

6.1.2 Evaluation Setup

To answer our research questions, we use the HSM2FSM transformation as our case study, and apply mutation analysis [26] to systematically inject faults. In particular, we specify 14 preconditions and 5 postconditions on the original HSM transformation from [10]. Then, we inject faults by applying a list of mutation operators defined in [9] on the transformation. We apply mutations only to the transformation because we focus on contract-based development, where the contract guides the development of the transformation. Our mutants are proved against the specified postconditions, and we apply our fault localization approach in case of unverified postconditions. We kindly refer to our online repository for the complete artifacts used in our evaluation[9].

6.1.3 Evaluation Results

Table 1 summarizes the evaluation results for our fault localization approach on the chosen case study. The first column lists the identity of the mutants[10]. The second and third columns record the unverified OCL postconditions and their corresponding verification time. The fourth, fifth, sixth and seventh columns record information of verifying sub-goals, i.e. the number of unverified sub-goals / total number of sub-goals (4th), average verification time of sub-goals (5th), the maximum verification time among sub-goals (6th), total verification of sub-goals (7th) respectively. The last column records whether the faulty lines (L_{faulty}, i.e. the lines that the mutation operators operated on) are presented in the problematic transformation scenarios (PTS) of unverified sub-goals.

[9] A deductive approach for fault localization in ATL MTs (Online). `https://github.com/veriatl/VeriATL/tree/FaultLoc`

[10] The naming convention for mutants are mutation operator Add(A) / Del(D) / Modify(M), followed by the mutation operand Rule(R) / Filter(F) / TargetElement(T) / Binding(B), followed by the position of the operand in the original transformation setting. For example, *MB1* stands for the mutant which modifies the binding in the first rule.

Table 1. Evaluation metrics for the *HSM2FSM* case study

	Unveri. Post.		Sub-goals				$L_{faulty} \in PTS$
	ID	Veri. Time(ms)	Unveri. / Total	Avg. Time (ms)	Max Time (ms)	Total Time (ms)	
MT2	#5	3116	3 / 4	1616	1644	6464	True
DB1	#5	2934	1 / 1	1546	1546	1546	-
MB6	#4	3239	1 / 12	1764	2550	21168	True
AF2	#4	3409	2 / 12	1793	2552	21516	True
MF6	#2	3779	0 / 6	1777	2093	10662	N/A
	#4	3790	1 / 12	1774	2549	21288	True
DR1	#1	2161	3 / 6	1547	1589	9282	-
	#2	2230	3 / 6	1642	1780	9852	-
AR	#1	3890	1 / 8	1612	1812	12896	True
	#3	4057	6 / 16	1769	1920	28304	True

First, we confirm that there is no inconclusive verification results of the generated sub-goals, i.e. if VeriATL reports that the verification result of a sub-goal is unverified, then it presents a fault in the transformation. Our confirmation is based on the manual inspection of each unverified sub-goal to see whether there is a counter-example to falsify the sub-goal. This supports the correctness of our fault localization approach. We find that the deduced hypotheses of the sub-goals are useful for the elaboration of a counter-example (e.g. when they imply that the fault is caused by missing code as the case in Listing 3).

Second, as we inject faults by mutation, identifying whether the faulty line is presented in the problematic transformation scenarios of unverified sub-goals is also a strong indication of the correctness of our approach. Shown by the last column, all cases satisfies the faulty lines inclusion criteria. 3 out 10 cases are special cases (dashed cells) where the faulty lines are deleted by the mutation operator (thus there are no faulty lines). In the case of *MF6#2*, there are no problematic transformation scenarios generated since all the sub-goals are verified. By inspection, we report that our approach improves the completeness of VeriATL. That is the postcondition (#2) is correct under *MF6* but unable to be verified by VeriATL, whereas all its generated sub-goals are verified.

Third, shown by the fourth column, in 5 out of 10 cases, the developer is presented with at most one problematic transformation scenario to pinpoint the fault. This positively supports the efficiency of our approach. The other 5 cases produce more sub-goals to examine. However, we find that in these cases each unverified sub-goal gives a unique phenomenon of the fault, which we believe is valuable to fix the bug. We also report that in rare cases more than one sub-goal could point to the same phenomenon of the fault. This is because the hypotheses of these sub-goals contain a semantically equivalent set of *genBy* predicates. Although they are easy to identify, we would like to investigate how to systematically filter these cases out in the future.

Fourth, from the third and fifth columns, we can see that each of the sub-goals is faster to verify than its corresponding postcondition by a factor of about 2. This is because we sent a simpler task than the input postcondition to verify, e.g. because of our transformation slicing, the VC for each sub-goal encodes a simpler interaction of transformation rules compared to the VC for its corresponding postcondition. From the third and sixth columns, we can further report that all sub-goals are verified in less time than their corresponding postcondition.

6.2 Scalability Evaluation

To evaluate the two steps we proposed for scalable MT verification, we first describe our research questions and the evaluation setup. Then, we detail the results of our evaluation.

6.2.1 Research questions

We formulate two research questions to evaluate the scalability of our verification approach:

(RQ1) Can a MT-specific slicing approach significantly increase verification efficiency w.r.t. domain-agnostic Boogie-level optimization when a MT is scaling up?

(RQ2) Can our proposed grouping algorithm improve over the slicing approach for large-scale MT verifications?

6.2.2 Evaluation Setup

To answer our research questions, we first focus on verifying a perfect UML copier transformation w.r.t. to the full set of 50 invariants (naturally we expect the copier to satisfy all the invariants). These invariants specify the well-formedness of UML constructs, similar to the ones defined in Listing 1. We implement the copier as an ATL MT that copies each classifier of the source metamodel into the target, and preserves their structural features (i.e. 194 copy rules). Note that while the copier MT has little usefulness in practice, it shares a clear structural similarity with real-world refactoring transformations. Hence, in terms of scalability analysis for deductive verification, we consider it to be a representative example for the class of refactoring transformations. We support this statement in Section 6.3, where we discuss the generalizability of our scalable approach by extending the experimentation to a set of real-world refactoring transformations.

Our evaluation consists of two settings, one for each research question. In the first setting, we investigate RQ1 by simulating a monotonically growing verification problem. We first sort the set of postconditions according to their verification time (obtained by verifying each postcondition separately before the experimentation). Then we construct an initial problem by taking the first (simplest) postcondition and the set of rules (extracted from the UML copier) that copy all the elements affecting the postcondition. Then we expand the problem by adding the next simplest postcondition and its corresponding rules, arriving after 50 steps to the full set of postconditions and the full UML copier transformation.

At each of the 50 steps, we evaluate the performance of 2 verification approaches:

- ORG_b. The original VeriATL verification system: each postcondition is separately verified using Boogie-level VC splitting.
- *SLICE*. Our MT slicing technique applied on top of the ORG_b approach: each postcondition is separately verified over the transformation slice impacting that specific postcondition (as described in Section 5.1).

Furthermore, we also applied our *SLICE* approach to a set of real-world transformations, to assess to which extent the previous results on the UML copier transformation are generalizable: we replaced the UML copier transformation in the previous experiment with 12 UML refactoring transformations from the ATL transformations zoo[11], and verified them against the same 50 OCL invariants. When the original UML refactorings contain currently non-supported constructs (please refer *language support* in Section 6.3 for details), we use our result on rule irrelevance (Theorem 2) to determine whether each invariant would produce the same VCs when applied to the copier transformation and to the refactorings. If not, we automatically issue *timeout* verification result to such invariant on the refactoring under study, which demonstrates the worst-case situation for our approach. By doing so, we ensure the fairness of the performance analysis for all the corpus.

For answering RQ2, we focus on the verification problem for the UML copier transformation, and compare two verification approaches, i.e. *SLICE* and *GROUP*, that ap-

[11] The ATL transformations zoo. `http://www.eclipse.org/atl/atlTransformations/`

plies the grouping algorithm on top of *SLICE* (as described in Section 5.2). In particular, we variate the pair of arguments MAX_t and MAX_s (i.e. maximum traces and subgoals per group) to investigate their correlation with the algorithm performance.

Our scalability evaluation is performed on an Intel 3 GHz machine with 16 GB of memory running the Linux operating system. We refer to our online repository for the complete artifacts used in our evaluation[12]

6.2.3 Evaluation Result

The two charts in Fig. 7 summarize the evaluation results of the first setting. In Fig. 7-(a) we record for each step the longest time taken for verifying a single postcondition at that step. In Fig. 7-(b) we record the total time taken to verify all the postconditions for each step. The two figures bear the same format. Their x-axis shows each of the steps in the first setting and the y-axis is the recorded time (in seconds) to verify each step by using the ORG_b and *SLICE* approaches. The grey horizontal line in Fig. 7-(a) shows the verifier timeout (180s).

We learn from Fig. 7-(a) that the *SLICE* approach is more resilient to the increasing complexity of the problem than the ORG_b approach. The figure shows that already at the 18th step the ORG_b approach is not able to verify the most complex postcondition (the highest verification time reaches the timeout). The *SLICE* technique is able to verify all postconditions in much bigger problems, and only at the 46th step one VC exceeds the timeout.

Moreover, the results in Fig. 7-(b) support a positive answer to RQ1. The *SLICE* approach consistently verifies postconditions more efficiently than the ORG_b approach. In our scenario the difference is significant. Up to step 18, both the approaches verify within timeout, but the verification time for ORG_b shows exponential growth while *SLICE* is quasi-linear. At step 18th, *SLICE* takes 11.8% of the time of ORG_b for the same verification result (171s against 1445s). For the rest of the experimentation ORG_b hits timeout for most postconditions, while *SLICE* loses linearity only when the most complex postconditions are taken into account (step 30).

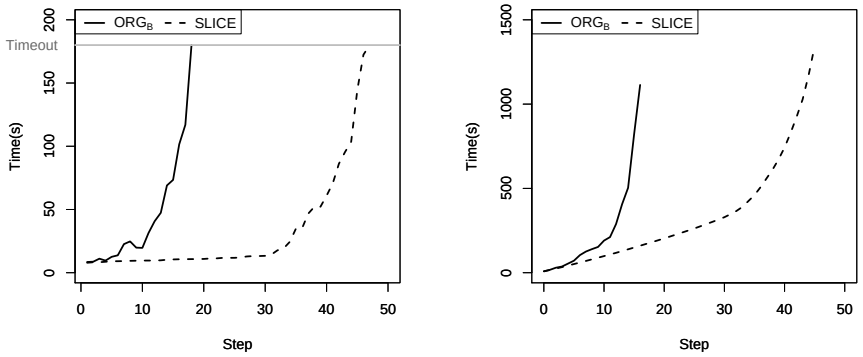
In our opinion, the major reason for the differences in shape of Fig. 7 is because the ORG_b approach always aligns postconditions to the whole set of transformation rules, whereas the *SLICE* approach aligns each postcondition only to the ATL rules it depends on, thereby greatly reducing the size of each constructed VC.

Table 2 shows to which extent the previous results on the UML copier transformation are generalizable to other MTs. For each transformation the table shows the verification time (in seconds) spent by the ORG_b and *SLICE* approaches respectively. The fourth column shows the improvement of the *SLICE* approach over ORG_b.

From Table 2, we learn that when using the *SLICE* approach on the corpus, on average 43 (50 - 7) out of 50 postconditions can expect a similar verification performance as observed in verifying the UML copier transformation. The reason is that our *SLICE* approach does not depend on the degree of supported features to align postconditions to the corresponding ATL rules. This gives more confidence that our approach can efficiently perform large scale verification tasks as shown in the previous experimentation, while we enable unsupported features.

We report that for the 12 transformations studied, the *SLICE* approach 1) is consistently faster than the ORG_b approach and 2) is consistently able to verify more postconditions than the ORG_b approach in the given timeout. On the full verification *SLICE* gains an average 71% time w.r.t. ORG_b. The most gain is in the *UML2Profiles* case, which we observe 78% speed up than ORG_b. The least gain is in the *UML2Java* case (68% speed up w.r.t. ORG_b), caused by 9 timeouts issued because of currently non-supported constructs (e.g. imperative call to helpers and certain iterators on OCL sequneces). All in all, these results con-

[12] On scalability of deductive verification for ATL MTs (Online). `https://github.com/veriatl/VeriATL/tree/Scalability`



(a) Max verification time for each step (b) Total verification time for each step

Fig. 7. The evaluation result of the first setting

Table 2. The generalization evaluation of the first setting

ID	**Time**$_{ORG_b}$	**Time**$_{SLICE}$	**Time Gained**
UMLCopier	9047	2065	77%
UML2Accessors	9094	2610	71%
UML2MIDlet	9084	2755	70%
UML2Profiles	9047	2118	77%
UML2Observer	9084	2755	70%
UML2Singleton	9094	2610	71%
UML2AsyncMethods	9084	2755	70%
UML2SWTApplication	9084	2755	70%
UML2Java	9076	2923	68%
UML2Applet	9094	2610	71%
UML2DataTypes	9014	2581	71%
UML2JavaObserver	9084	2755	70%
UML2AbstractFactory	9094	2610	71%
Average	*9078*	*2653*	*71%*

firm the behavior observed in verifying the UML copier transformation.

Table 3 shows the evaluation result of the second setting. The first two columns record the two arguments sent to our grouping algorithm. In the 3rd column, we calculate the *group ratio* (GR), which measures how many of the 50 postconditions under verification are grouped with at least another one by our algorithm. In the 4th column (*success rate*), we calculate how many of the grouped VCs are in groups that decrease the global verification time. Precisely, if a VC P is the grouping result of VCs P_1 to P_n, T_a is the verification time of P using the *GROUP* approach, T_2 is the sum of the verification times of P_1 to P_n using the *SLICE* approach, then we consider P_1 to P_n are successfully grouped if T_1 is not reaching timeout and T_1 is less than T_2. In the 5th column, we record the speedup ratio (SR), i.e. the difference of global verification time between the two approaches divided by the global verification time of the *SLICE* approach. In the 6th column, we record the

Table 3. The evaluation result of the second setting

$\mathbf{Max}_t$	$\mathbf{Max}_s$	**GR**	**Succ. Rate**	**SR**	**TS**
3	10	8%	100%	48%	16
4	13	22%	100%	49%	51
5	15	44%	100%	47%	108
6	18	50%	100%	51%	134
7	20	56%	93%	23%	73
8	23	62%	81%	11%	41
9	25	64%	72%	-108%	-400
10	28	62%	55%	-158%	-565
11	30	64%	31%	-213%	-789
12	33	68%	0%	-212%	-1119
13	35	68%	18%	-274%	-1445
14	38	72%	17%	-433%	-3211
15	40	72%	0%	-438%	-3251
16	43	76%	0%	-547%	-4400
17	45	76%	0%	-620%	-4988

time saved (TS) by the *GROUP* approach, by calculating the difference of global verification time (in seconds) between the two approaches.

The second setting indicates that our grouping algorithm can contribute to performance on top of the slicing approach, when the parameters are correctly identified. In our evaluation, the highest gain in verification time (134 seconds) is achieved when limiting groups to 6 maximum traces and 18 subgoals. In this case, 25 VCs participate in grouping, all of them successfully grouped. Moreover, we report that these 25 VCs would take 265 seconds to verify by using the *SLICE* approach, more than twice of the time taken by the *GROUP* approach. Consequently, the *GROUP* approach takes 1931 seconds to verify all the 50 VCs, 10% faster than the *SLICE* approach (2065 seconds), and 79% faster than the ORG_b approach (9047 seconds).

Table 3 also shows that the two parameters chosen as arguments have a clear correlation with the grouping ratio and success rate of grouping. When the input arguments are gradually increased, the grouping ratio increases (more groups can be formed), whereas the success rate of grouping generally decreases (as the grouped VCs tend to become more and more complex). The effect on verification time is the combination of these two opposite behaviors, resulting in a global maximum gain point (MAX_t=6, MAX_s=18).

Finally, Table 3 shows that the best case for grouping is obtained by parameter values that extend the group ratio as much as possible, without incurring in a loss of success rate. However, the optimal arguments for the grouping algorithm may depend on the structure of the transformation and constraints. Their precise estimation by statically derived information is an open problem, that we consider for future work. Table 3 and our experience have shown that small values for the parameters (like in the first 5 rows) are safe pragmatic choices.

6.3 Discussions

In summary, our evaluations give a positive answer to all of our four research questions. It confirms that our fault localization approach can correctly and efficiently pinpoint the faults in the given MT: (a) faulty constructs are presented in the sliced transformation; (b) deduced clues assist developers in various debugging tasks (e.g. the elaboration of a counter-example); (c) the number of sub-goals that need to be examined to pinpoint a fault is usually small. Moreover, our scalability evaluation shows that our slicing and algorithmic VC grouping approaches improve verification performance up to 79% when a MT is scaling up. However, there are also lessons we learned from the two evaluations.

Completeness. We identify three sources of incompleteness w.r.t. our proposed approaches.

First, incomplete application of the automated proof strategy (defined in Definition 1). Clearly, if not detected, an incomplete application of our automated proof strategy could cause our transformation slicing to erroneously slice away the rules that a postcondition might depend on. In our current solution we are able to detect incomplete cases, report them to the user, and defensively verify them. We detect incomplete cases by checking whether every elements of target types referred by each post-

condition are accompanied by a *genBy* predicate (this indicates full derivation). While this situation was not observed during our experimentation, we plan to improve the completeness of the automated proof strategy in future by extending the set of natural deduction rules for ATL and design smarter proof strategies. By defensive verification, we mean that we will construct the slice to be the full MT for the incomplete cases. Thus, the VCs of the incomplete cases become MM, Pre, Exec $\vdash$ Post, and fault localization is automatically disabled in these cases.

Second, incomplete verification. The Boogie verifier may report inconclusive results in general due to the underlying SMT solver. We hope the simplicity offered by our fault localization approach would facilitate the user in making the distinction between incorrect and inconclusive results. In addition, if the verification result is inconclusive, our fault localization approach can help users in eliminating verified cases and find the source of its inconclusiveness. In the long run, we plan to improve completeness of verification by integrating our approaches to interactive theorem provers such as Coq [7] and Rodin [2] (e.g. drawing on recursive inductive reasoning). One of the easiest paths is exploiting the Why3 language [19], which targets multiple theorem provers as its back-ends.

Third, incomplete grouping. The major limitation of our grouping algorithm is that we currently have not proposed any reliable deductive estimation of optimal parameters MAX_t and MAX_s for a given transformation. Our evaluation suggests that conservatively choosing these parameters could be a safe pragmatic choice. Our future work would be toward more precise estimation by integrating with more statically derived information.

Generalization of the experimentation. While evaluating our fault localization approach, we take a popular assumption in the fault localization community that multiple faults perform independently [44]. Thus, such assumption allows us to evaluate our fault localization approach in a one-post-condition-at-a-time manner. However, we cannot guarantee that this is general for realistic and industrial MTs. We think classifying contracts into related groups could improve these situations.

To further improve the generalization of our proposed approaches, we also plan to use synthesis techniques to automatically create more comprehensive contract-based MT settings. For example, using metamodels or OCL constraints to synthesize consistency-preserving MT rules [28, 36], or using a MT with OCL postconditions to synthesize OCL preconditions [18].

Language Support. Our implementation supports a core subset of the ATL and OCL languages: (a) declarative ATL (matched rules) in non-refining mode, many-to-many mappings of (possibly abstract) classifiers with the default resolution algorithm of ATL; (b) first-order OCL contracts, i.e. OCL-Type, OCLAny, Primitives (OCLBool, OCLInteger, OCLString), Collection data types (i.e. Set, OrderedSet, Sequence, Bag), and 78 OCL operations on data types, including the *forAll*, *collect*, *select*, and *reject* iterators on collections. Refining mode (that uses in-place scheduling) is supported by integrating our previous work [14]. The imperative and recursive aspects of ATL are currently not considered.

Usability. Currently, our fault localization approach relies on the experience of the transformation developer to interpret the deduced debugging clues. We think that counter-example generation would make this process more user-friendly, e.g. like quickcheck in Haskell [16], or random testing in Isabelle/HOL [5]. In [17], the authors show how to combine derived constraints with model finder to generate counter-examples that uncover type errors in MTs. In the future, we plan to investigate how to use this idea to combine our debugging clues with model finders to ease the counter-example generation in our context.

Finally, in case of large slices, we plan to automatically prioritize which unverified sub-goals the user needs to examine first (e.g. by giving higher priority to groups of unverified sub-goals within the same branch of the proof tree). We are also working in eliminating sub-

goals that are logically equivalent (as discussed in Section 6.1.3).

7 Related Work

Scalable Verification of MT. There is a large body of work on the topic of ensuring MT correctness [1], or program correctness in general [21, 35].

Poernomo outlines a general proof-as-model-transformation methodology to develop correct MTs [34]. The MT and its contracts are first encoded in a theorem prover. Then, upon proving them, a functional program can be extracted to represent the MT based on the Curry-Howard correspondence [24].

UML-RSDS is a tool-set for developing correct MTs by construction [29]. It chooses well-accepted concepts in MDE to make their approach more accessible by developers, i.e. it uses a combination of UML and OCL to create a MT design and contracts.

Calegari et al. encode the ATL MTr and its metamodels into inductive types [12]. The contracts for semantic correctness are given by OCL, which are translated into logical predicates. As a result, they can use the Coq proof assistant to interactively verify that the MTr is able to produce target models that satisfy the given contracts

Büttner et al. use Z3 to verify a declarative subset of the ATL and OCL contracts [10]. Their approach aims at providing minimal axioms that can verify the given OCL contracts.

Our work complements these works by focusing on scalability to make the verification more practical. To our knowledge our proposal is the first applying transformation slicing to increase the scalability of MT verification. Our work is close to Leino et al. [30]. They introduce a Boogie-level VC splitting approach based on control-flow information. For example, the **then** and else blocks of an **if** statement branch the execution path, and can be hints for splitting VCs. This optimization does not have significant results in our context because the control-flow of ATL transformations is simple, yielding a single execution path with no potential to be split. This motivated us to investigate language-specific VC optimizations based on static information of ATL transformations. Our evaluation shows the integration of the two approaches is successful.

Fault Localization. Being one of the most user-friendly solutions to provide the users with easily accessible feedback, partially or fully automated fault localization has drawn a great attention of researchers in recent years [37,44]. Program slicing refers to identification of a set of program statements which could affect the values of interest [40, 43], and is often used for fault localization of general programming languages. W.r.t. other program slicing techniques, our work is more akin to traditional statement-deletion style slicing techniques than the family of amorphous slicing [20], since our approach does not alter the syntax of the MT for smaller slices. While amorphous slicing could potentially produce thinner slices for large MTs (which is important for the practicability of verification), we do not consider it in this work because: (a) the syntax-preserving slices constructed by the traditional approach is a more intuitive information to debug the original MT; (b) the construction of an amorphous slice is more difficult, since to ensure correctness, each altered part has to preserve the semantics of its correspondence.

Few works have adapted the idea of program slicing to localize faults in MTs. Aranega et al. define a framework to record the runtime traces between rules and the target elements these rules generated [3]. When a target element is generated with an unexpected value, the transformation slices generated from the run-time traces are used for fault localization. While Aranega et al. focus on dynamic slicing, our work focuses on static slicing which does not require test suites to exercise the transformation.

To find the root of the unverified contracts, Büttner et al. demonstrate the UML-2Alloy tool that draws on the Alloy model finder to generate counter examples [11]. However, their tool does not guarantee that the newly generated counter example gives additional information than the previous ones. Oakes et

al. statically verify ATL MTs by symbolic execution using DSLTrans [32]. This approach enumerates all the possible states of the ATL transformation. If a rule is the root of a fault, all the states that involve the rule are reported.

Sánchez Cuadrado et al. present a static approach to uncover various typing errors in ATL MTs [17], and use the USE constraint solver to compute an input model as a witness for each error. Compared to their work, we focus on contract errors, and provide the user with sliced MTs and modularized contracts to debug the incorrect MTs.

The most similar approach to ours is the work of Burgueño et al. on syntactically calculating the intersection constructs used by the rules and contracts [9]. To our knowledge our proposal is the first applying natural deduction with program slicing to increase the precision of fault localization in MT. W.r.t. the approach of Burgueño et al., we aim at improving the localization precision by considering also semantic relations between rules and contracts. This allows us to produce smaller slices by semantically eliminating unrelated rules from each scenario. Moreover, we provide debugging clues to help the user better understand why the sliced transformation causing the fault. However, their work considers a larger set of ATL. We believe that the two approaches complement each other and integrating them is useful and necessary.

8 Conclusion and Future Work

In summary, in this work we confronted the fault localization and scalability problems for deductive verification of MT. In terms of the fault localization problem, we developed an automated proof strategy to apply a set of designed natural deduction rules on the input OCL postcondition to generate sub-goals. Each unverified sub-goal yields a sliced transformation context and debugging clues to help the transformation developer pinpoint the fault in the input MT. Our evaluation with mutation analysis positively supports the correctness and efficiency of our fault localization approach. The result showed that: (a) faulty constructs are presented in the sliced transformation, (b) deduced clues assist developers in various debugging tasks (e.g. to derive counter-examples), (c) the number of sub-goals that need to be examined to pinpoint a fault are usually small.

In terms of scalability, we lift our slicing approach to postconditions to manage large scale MTs by aligning each postcondition to the ATL rules it depends on, thereby reducing the verification complexity/time of individual postcondition. Moreover, we propose and prove a grouping algorithm to identify the postconditions that should be compositionally verified to improve the global verification performance. Our evaluation confirms that our approach improves verification performance up to 79% when a MT is scaling up.

Our future work includes facing the limitations identified during the evaluation (Section 6.3). We also plan to extend our slicing approach to metamodels and preconditions, i.e. slicing away metamodel constraints or preconditions that are irrelevant to each sub-goal. This would allow us to further reduce the size of problematic transformation scenario for the users to debug faulty MTs.

In addition, we plan to investigate how our decomposition can help us in reusing proof efforts. Specifically, due to requirements evolution, the MT and contracts are under unpredictable changes during the development. These changes can invalidate all of the previous proof efforts and cause long proofs to be recomputed. We think that our decomposition of sub-goals would increase the chances of reusing verification results, i.e. sub-goals that are not affected by the changes.

References

1. Ab.Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Software & Systems Modeling 14(2), 1003–1028 (2015)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An open toolset for modelling and reasoning

in Event-B. International Journal on Software Tools for Technology Transfer 12(6), 447–466 (2010)
3. Aranega, V., Mottu, J., Etien, A., Dekeyser, J.: Traceability mechanism for error localization in model transformation. In: 4th International Conference on Software and Data Technologies. pp. 66–73. Sofia, Bulgaria (2009)
4. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: 4th International Conference on Formal Methods for Components and Objects. pp. 364–387. Springer, Amsterdam, Netherlands (2006)
5. Berghofer, S., Nipkow, T.: Random Testing in Isabelle/HOL. In: 2nd International Conference on Software Engineering and Formal Methods. pp. 230–239. IEEE, Beijing, China (2004)
6. Berry, G.: Synchronous design and verification of critical embedded systems using SCADE and Esterel. In: 12th International Workshop on Formal Methods for Industrial Critical Systems, pp. 2–2. Springer, Berlin, Germany (2008)
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. Springer, 1st edn. (2010)
8. Briand, L.C.: Making model-driven verification practical and scalable - experiences and lessons learned. In: 4th International Conference on Model-Driven Engineering and Software Development. SCITEPRESS, Rome, Italy (2016)
9. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Static fault localization in model transformations. IEEE Transactions on Software Engineering 41(5), 490–506 (2015)
10. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: 15th International Conference on Model Driven Engineering Languages and Systems. pp. 198–213. Springer, Innsbruck, Austria (2012)
11. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: 14th International Conference on Formal Engineering Methods. pp. 198–213. Springer, Kyoto, Japan (2012)
12. Calegari, D., Luna, C., Szasz, N., Tasistro, Á.: A type-theoretic framework for certified model transformations. In: 13th Brazilian Symposium on Formal Methods. pp. 112–127. Springer, Natal, Brazil (2011)
13. Cheng, Z., Monahan, R., Power, J.F.: A sound execution semantics for ATL via translation validation. In: 8th International Conference on Model Transformation. pp. 133–148. Springer, L'Aquila, Italy (2015)
14. Cheng, Z., Monahan, R., Power, J.F.: Formalised EMFTVM bytecode language for sound verification of model transformations. Software & Systems Modeling p. In Press (2016)
15. Cheng, Z., Tisi, M.: A deductive approach for fault localization in ATL model transformations. In: 20th International Conference on Fundamental Approaches to Software Engineering. pp. 300–317. Springer, Uppsala, Sweden (2017)
16. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ACM SIGPLAN Notices 46(4), 53–64 (May 2011)
17. Cuadrado, J.S., Guerra, E., de Lara, J.: Uncovering errors in ATL model transformations using static analysis and constraint solving. In: 25th IEEE International Symposium on Software Reliability Engineering. pp. 34–44. IEEE, Naples, Italy (2014)
18. Cuadrado, J.S., Guerra, E., de Lara, J., Clarisó, R., Cabot, J.: Translating target to source constraints in model-to-model transformations. In: 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 12–22. IEEE, Austin, TX, USA (2017)
19. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: 22nd European Symposium on Programming. pp. 125–128. Springer, Rome, Italy (2013)
20. Harman, M., Binkley, D., Danicic, S.: Amorphous program slicing. Journal of Systems and Software 68(1), 45 – 64 (2003)
21. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.J.: Behavioral interface specification languages. ACM Computing Surveys 44(3), 16:1–16:58 (2012)
22. Hidaka, S., Jouault, F., Tisi, M.: On additivity in transformation languages. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems. pp. 23–33. IEEE, Austin, TX, USA (2017)

23. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
24. Howard, W.A.: The formulae-as-types notion of construction. In: To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 479–490. Academic Press (1980)
25. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press (2004)
26. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (2011)
27. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
28. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: 9th International Conference on Model Transformation. pp. 173–188. Springer, Vienna, Austria (2016)
29. Lano, K., Clark, T., Kolahdouz-Rahimi, S.: A framework for model transformation verification. Formal Aspects of Computing 27(1), 193–235 (2014)
30. Leino, K.R.M., Moskal, M., Schulte, W.: Verification condition splitting. `https://www.microsoft.com/en-us/research/publication/verification-condition-splitting/` (2008)
31. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer, Budapest, Hungary (2008)
32. Oakes, B.J., Troya, J., Lúcio, L., Wimmer, M.: Fully verifying transformation contracts for declarative ATL. In: 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 256–265. IEEE, Ottawa, ON (2015)
33. Object Management Group: Unified modeling language (ver. 2.5). `http://www.omg.org/spec/UML/2.5/` (2015)
34. Poernomo, I.: Proofs-as-model-transformations. In: 1st International Conference on Model Transformation. pp. 214–228. Springer, Zürich, Switzerland (2008)
35. Prasad, M.R., Biere, A., Gupta, A.: A survey of recent advances in SAT-based formal verification. International Journal on Software Tools for Technology Transfer 7(2), 156–173 (2005)
36. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential OCL invariants to nested graph constraints focusing on set operations. In: 8th International Conference on Graph Transformation. pp. 155–170. Springer, L'Aquila, Italy (2015)
37. Roychoudhury, A., Chandra, S.: Formula-based software debugging. Communications of the ACM pp. 68–77 (2016)
38. Selim, G., Wang, S., Cordy, J., Dingel, J.: Model transformations for migrating legacy models: An industrial case study. In: 8th European Conference on Modelling Foundations and Applications. pp. 90–101. Springer, Lyngby, Denmark (2012)
39. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse modeling framework. Pearson Education, 2nd edn. (2008)
40. Tip, F.: A survey of program slicing techniques. Tech. rep., Centrum Wiskunde & Informatica (1994)
41. Tisi, M., Perez, S.M., Choura, H.: Parallel execution of ATL transformation rules. In: 16th International Conference on Model-Driven Engineering Languages and Systems. pp. 656–672. Springer, Miami, FL, USA (2013)
42. Wagelaar, D.: Using ATL/EMFTVM for import/export of medical data. In: 2nd Software Development Automation Conference. Amsterdam, Netherlands (2014)
43. Weiser, M.: Program slicing. In: 5th International Conference on Software Engineering. pp. 439–449. IEEE, NJ, USA (1981)
44. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Transactions on Software Engineering Pre-Print(99), 1–41 (2016)